Implementation of the High-level Parallel Programming Language Nano-2

AUTHOR(S):

HIRABARU, Masaki; ARAKI, Keijiro; SUEYOSHI, Toshinori; ARITA, Itsujiro

数理解析研究所講究録
第 586 巻 1986 年 112-133

Implementation of the High-level Parallel Programming Language Nano-2

Masaki HIRABARU† (平原 正樹)
Keijiro ARAKI† (荒木 啓二郎)
Toshinori SUEYOSHI† (末吉 敏則)
Itsujiro ARITA†† (有田 五次郎)

†Kyushu University (九州大学)
††Kyushu Institute of Technology (九州工業大学)

Abstract

The high-level parallel programming language Nano-2[3] was designed for shared memory multiprocessors and supports several features to write reliable parallel programs. The language Nano-2 was implemented on a prototype of highly parallel processing system[10], HYPHEN C-16[15]. This paper describes the language Nano-2 and shows that the parallel features provided by this language can be efficiently implemented on the shared memory multiprocessor. The merits of the programming system of Nano-2 is also discussed.

1. Introduction

The evolution of hardware technology has made it feasible to built multiprocessor systems with a large number of processors. HYPHEN C-16[15], a prototype of such multiprocessor systems, was constructed in 1982 and consists of sixteen microprocessor modules with its own shared memory. On this multiprocessor system, we had implemented the following programming systems:

(1)the programming system for performance evaluation[16] which consists of compilers of Fortran and Pascal, an assembler, etc., and

(2)the programming system of the parallel programming language P-Pascal[9] which is based on Pascal.

In the first system, the run-time efficiency of programs took precedence over easiness of programming since the performance evaluation of its hardware was the main purpose then. Therefore it was needed to closely know HYPHEN C-16 and compelling to adapt the sequential language to parallel programming. In the second system, we designed a parallel programming language based on the sequential Pascal and implemented it. It surely provides high-level features in sequential programming but it is still insufficient in parallel programming.

It is said to be hard to write parallel programs and to confirm their behavior, which have really known from our experience. In contrast to most of parallel systems (including our two systems above) which intend to speed up only execution of programs, this research intends to decrease total cost of programming as well as execution. Without programming environment, it is difficult to attain the maximum performance of high-speed parallel computers.

In order to utilize multiprocessor systems and improve software reliability, a high-level parallel programming language is indispensable. Most of existing concurrent programming languages, however, have designed to provide simulated parallelism on a single processor. There is no guarantee that such a language can be efficiently implemented on a multiprocessor. We have designed the high-level parallel programming language Nano-2 and implemented it on HYPHEN C-16. The implementation and programming experience in the real parallel environment has led to the revision of Nano-2. We have implemented the revised version of Nano-2.

In this paper, we describe the revised version of the language Nano-2 and the implementation of its processing system and distributed run-time routines. In section 2 we discuss the design principles of Nano-2. In section 3 we briefly describe the parallel futures of Nano-2 with simple examples. In section 4 we describe an overview of HYPHEN C-16 and the implementation of Nano-2 on it. Concluding remarks are given in section 5.

2. Design Principles of Nano-2

Nano-2 was designed to facilitate the construction of parallel programs which run on the HYPHEN system. We did not intend

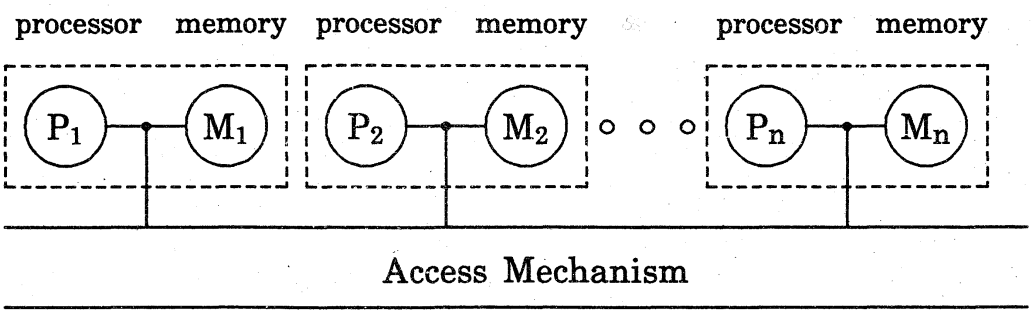

Fig.1 Model of target parallel computers

merely to add mechanisms for inter-process communication and synchronization to an existing language such as Pascal or C, but decided to design a new parallel programming language to support structured programming, information hiding, readability, etc. We took the following considerations into account in designing Nano-2.

(1)Nano-2 is a high-level parallel programming language for writing programs which run on a shared memory multiprocessor such as the HYPHEN system. Fig.1 shows a model of target parallel computers. Each Memory is accessed from the directly connected processor as a local memory, and also accessed from the others through the access mechanism. That is, each memory plays both roles of a local memory and a global one. Languages and systems for the model must provide facilities of partitioning and allocation as well as interaction between partitions.

(2)Nano-2 supports high modularity. In Nano-2, a *module* serves an abstract construct which is invoked and runs in parallel with the invoker. A module itself is also a parallel construct consisting of some other modules. Thus a Nano-2 program is hierarchically structured, but does not have Pascal-like nested block structures. The hierarchical structure with the module constructs supports abstraction of processing and isolation of machine dependencies. Nano-2 belongs to the group exemplified by Concurrent Pascal[7] and Modula[17]. Languages of this group provide parallel constructs as passive entities and also provide a parallel invocation mechanism for them. Languages of another group, exemplified by Concurrent Euclid[12] and Ada[1]†, provide parallel constructs

†Ada is a registered trademark of the U.S. Government-Ada Joint Program Office.

called process or task as active entities. Parallel constructs of Nano-2 called *task* are invoked by a mechanism of parallel invocation, called *parallel procedure call with reply*.

(3)All constructs in Nano-2 have closed scope. Explicit control over name visibility via import and export declarations is inherited from the programming language Euclid[13]. Names must be explicitly imported (or exported) via these declarations. This kind of information expressed explicitly in the program text is valuable for understanding and maintenance of the program. Accessibility is also controlled in the same declarations. Thus Nano-2 helps reliable programming. For example, Nano-2 provides shared variables, and it is important to control visibility and accessibility of them. By extracting many information from a source text such as type, import or export declarations, the compiler will do very strict checks of type, visibility and accessibility. Of course Nano-2 provides a facility of separate compilation to improve software productivity.

Although Nano-2 draws much on Euclid, we did not take account of verification of Nano-2. Reliability of programs on multiprocessor systems, such as fault tolerance, is also not included.

3. Language Features of Nano-2

Experience with the previous version of Nano-2 have brought to revise it. The following outlines language features of the revised version with example programs which really ran on HYPHEN C-16.

3.1 Simple Example

First we explain some of the language features in a simple example shown in Fig.2. The summation of 1 through 4 is divided into two parts: a subtotal of 1 and 2, and the other subtotal of 3 and 4, which are calculated on different processors in parallel. The node with two branching arrows corresponds to a parallel invocation, while the confluence corresponds to a synchronization. Fig.3 shows this program written in Nano-2, and the following gives an outline of it.

(1)The program begins with invoking task T1234 of module Add1234.

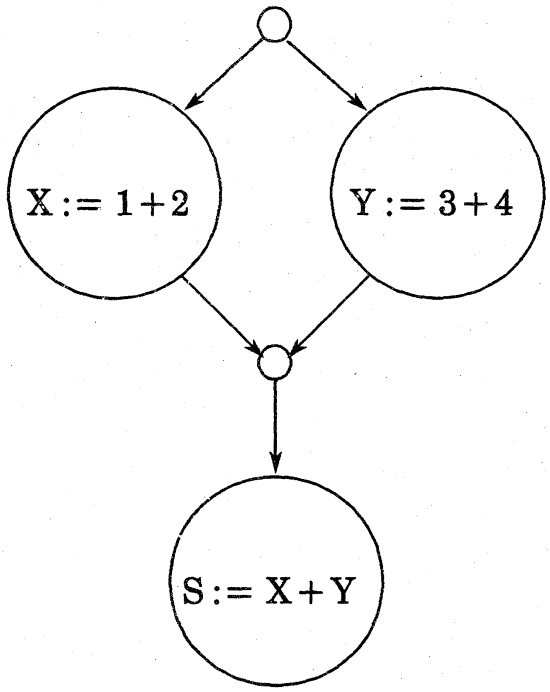

Fig.2 Simple example of summation

(2)Successively the task T1234 executes the *paracall* statement which invokes task T12 of module M12 and task T34 of module M34 at the same time, and waits for replies from both the tasks.

(3)Each invoked task executes the adding operation in parallel with the other and reaches the reply statement represented by *!!*. The statement returns a reply as an acknowledgment that the subtotal has already assigned to the shared variable.

(4)Receiving both of the acknowledgments, the invoker task T1234 resumes its execution to get a final result.

—Program partitioning—

Fig.4 shows the above program partitioning. All the entities declared in the same module, except floating constructs (function Add in Fig.3), must be allocated into the same processor. The function Add is referred from every module. In order to increase the amount of parallelism this function had better be individually executed in parallel on the each processor. The keyword *float* in the function declaration specifies making it floating. An invocation of a normal (i. e., non-floating) construct implies remote execution, while an

```
module Add1234; -- this module adds 1,2,3 and 4
  exports T1234; -- exports the entry of this module into outside

  function float Add (x, y: integer) returns integer;
  begin
    return x+y;
  end Add;

  module M12;
    imports Add; -- imports function Add
    exports X,T12; -- exports the result variable and entry
    var X: integer;

    task T12;
      imports Add, var X; -- imports X for variable
    begin
      X := Add (1, 2); !!; -- sets a subtotal then returns a reply
    end T12;

  end M12;

  module M34; -- same as M12
    imports Add;
    exports Y, T34;
    var Y: integer;

    task T34;
      imports Add, var Y;
    begin
      Y := Add (3, 4); !!;
    end T34;

  end M34;

  var S: integer;
  task T1234;
    imports Add, M12, M34, var S; -- imports sub-modules
  begin
    paracall M12.T12, M34.T34; -- invokes two tasks and waits
    S := Add (M12.X, M34.Y); -- gets a final result
    !!; -- returns a reply to the system
  end T1234;

end Add1234;-- this program is just a example.
```

Fig.3 Simple example written in Nano-2

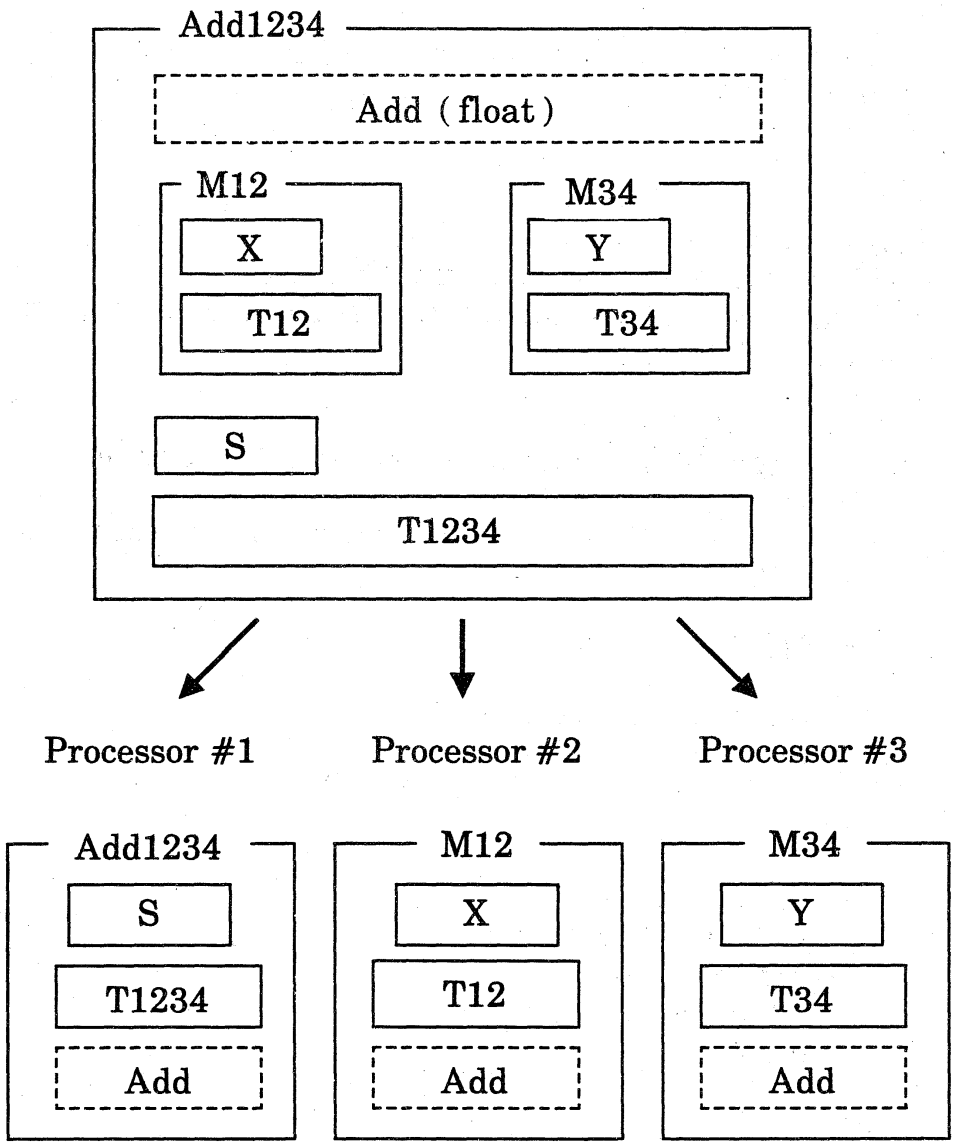

Fig.4 Partitioning of the program in Fig.3

invocation of a floating construct implies local execution by the caller. There is, however, no difference between them from the logical viewpoint of execution.

A unit of partitioning is a combination between a normal module and all floating constructs referred from it and is regarded as a logical processor (the processor number in Fig.4 is logical). We write a program in Nano-2 considering its partitioning, but it is little different from modular programming considering its functional specification. In

this current version, a program can be allocated only statically. Dynamic allocation is a further problem.

—Visibility control and hierarchical structure—

A construct in Nano-2 has closed scope. Visibility of names is explicitly controlled in much the same way as in Euclid. A name is visible in the scope in which it is declared. If it is to be visible in the contained scopes then it must be explicitly imported into those scopes via an import declaration. Names declared in a construct are visible outside of it if and only if they are explicitly exported from it into the enclosing construct by an export declaration. Accessibility is also controlled in the same declarations. Variables imported (or exported) with a keyword *var* are assignable. The use of export declarations are currently limited in only module constructs. These control will increase reliability of programs which use shared variables.

In the example, module Add1234 includes module M12 and M34 which may be allocated to different processors and executed in parallel. Outside of module Add1234, only visible is task T1234 as the entry of this module. The invocation of this entry looks like processing on a single virtual processor.

—Interaction between constructs—

We had intended to describe SPPs (Self-synchronizing Parallel Programs)[4] in the previous version of Nano-2. Although SPPs are parallel programs with high-speed synchronizing mechanism using hardware FIFO queues, abstraction of processing can be hardly archived in SPPs.

Now, a new synchronizing mechanism, which is called *parallel procedure call with reply*, is introduced in the current version of Nano-2. Fig.5 illustrates this mechanism. A parallel procedure call may invoke multiple tasks with parameters at the same time. The invoker task is blocked until receiving replies from all invoked tasks. Every invoked task may return a reply at any time. It may be regarded as the Dijkstra's *cobegin-coend* statement with the restriction that every statement included in it is only a invocation of a task (i.e. so-called remote procedure call). Varying the period of synchronization by

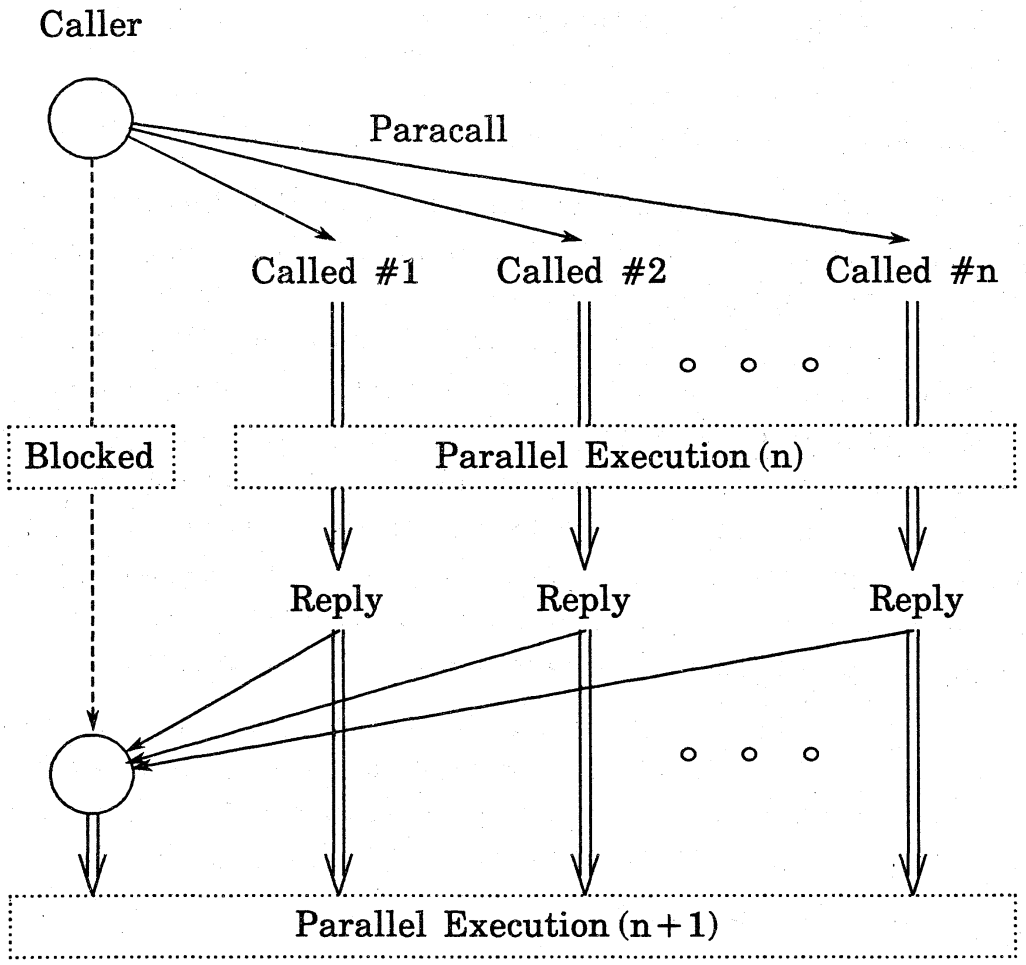

Fig.5 Mechanism of parallel procedure call with reply

returning a reply in an arbitrary place may be equivalent to a function of a disconnect instruction in the language Communication Port[14].

Nano-2 also provides *remote procedure call* as another invocation mechanism. Procedures and functions even in other modules can be called in a ordinary manner. Calling them allocated the same module means local procedure (or function) call, while calling them allocated other modules means remote procedure (or function) call. Therefore we do not need take account of partitioning when using the mechanism of procedure (or function) call. Since tasks, procedures and functions in a Nano-2 program must be implemented as reentrant, they can recursively invoke their own entries. Every invocation mechanism, parallel procedure call, procedure call and function call are little different from each other. Consequently, it will be a natural extension

from sequential programming to write a recursive parallel program with parallel procedure calls and/or remote procedure calls.

Of course, using invocations with parameters, the example shown in Fig. 2 can be rewritten not to use shared variables.

3.2 Module Array

A program module executed on each processor may be much the same as the others in some applications. Writing each program module individually would not only be a laborious work but also make the program hard to understand. *Module array* is introduced to solve these problems.

—Examples—

```
module Multiply (n);
                o
                o
                o
  task Times (a, b: column; var c: column);
  loop
     -- calculation
    paracall Multiply [Multiply' index mod n+1].Rotate (b);
  end loop;

  task Rotate (b: column);

                o
                o
                o
for i := 1 to n paracall Multiply [i].Times (a[i], b[i], c[i]);
```

Fig.6 Program of matrix multiplication

Fig.6 partly shows a program for multiplying two matrices, whose control flow is shown in Fig.7. The declaration, module Multiply(n), (where n is constant) specifies that it is a module array with n elements. The expression, Multiply'index, returns a value of its own

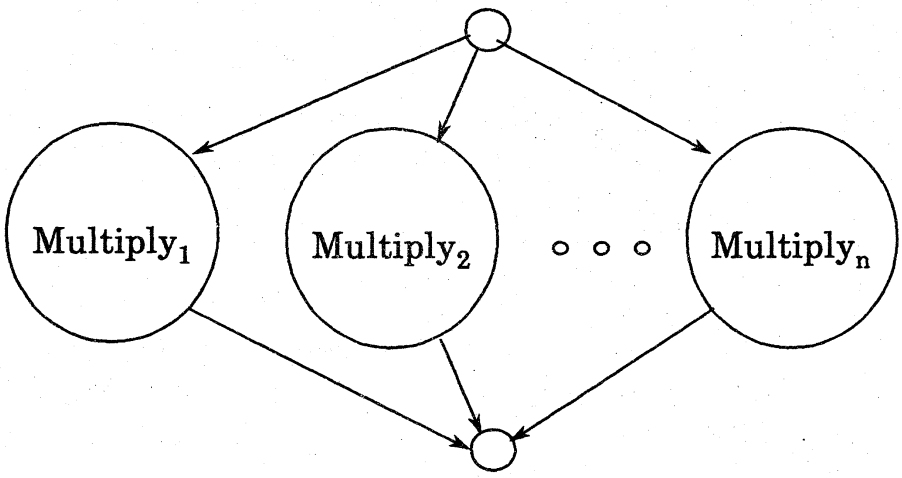

Fig.7 Control flow of matrix multiplication

position, thus every module can know its own position in the module array. The task Times of i-th module of Multiply receives column data, a[i] and b[i], and calculates the inner product by communicating to an adjacent, and sets a result to c[i]. With module arrays we could easily write this kind of programs.

```
module Sieve (n);
    o
    o
    o
  task Test (p: integer);
  -- if p passes this sieve
       paracall Sieve [Sieve' index + 1].Test (p);
    o
    o
    o
for i := 1 to x paracall Sieve[1].Test (i);
```

Fig.8 Program of Eratosthenes's sieve

Fig.8 shows another example, a part of program of the Eratosthenes's sieve, whose control flow is shown in Fig.9. It is a typical example of pipeline processing. The task Test of each module of Sieve receives a number p and tests whether p is caught in this sieve or not. If not, p is sent to the task of the next module.

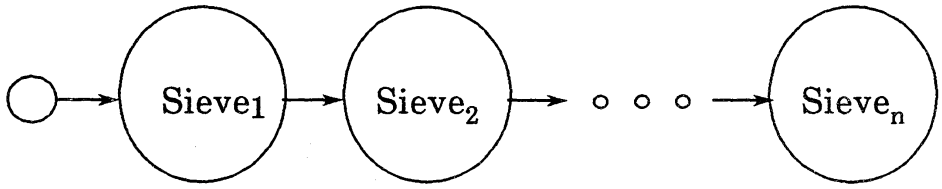

Fig.9 Control flow of Eratosthenes's sieve

—Implementation strategy—

The previous version of Nano-2 provided the above facility as a repetitive compilation of the compiler control facility[2] which was implemented in the preprocessor to expand a module array into the source text. This strategy increases the amount of both processing time and space in proportion to the number of elements, i.e., the number of processors. This situation is not agreeable because the language Nano-2 was designed for computers which consist of a large number of processors. The revised version provides this facility as a part of the language specification, and it becomes possible to defer an expansion of a module array until loading its object into every processor.

3.3 Mutual Exclusion

A language for parallel programming must provide the means to guard against time-dependent errors. Such errors can occur by the use of either shared variables or parallel invocations. Instead of completely eliminating time-dependent errors, we provide the language construct which will reduce them.

In SPPs of the previous version of Nano-2, switching a processor to another task could occur only at the end of the execution of a task. That is, code sequences in tasks of Nano-2 had to be continuously executed and were essentially critical regions. SPPs provide only this simple critical regions for mutual exclusion, but no general means with conditions such as guarded command[8]. As we have hidden the concept of SPPs by parallel procedure call with reply, we will also introduce a high-level language construct for mutual exclusion, *monitor*[11] with *signal* and *wait* operations over condition variables.

In a monitor, only one task can be executed at a time and synchronization conditions can be written by using signal and wait operations.

Obviously, the monitor construct may reduce the amount of parallelism for the sake of safety. For that cases, Nano-2 also provides *semaphore* as another mechanism for synchronization.

4. Implementation of Nano-2

We have implemented Nano-2 on the HYPHEN C-16 multiprocessor system. This section describes the implementation details.

4.1 Hardware Structure

HYPHEN C-16 is a prototype machine consisting of sixteen 8-bit microprocessors. Fig.10 shows the current configuration of HYPHEN C-16 connected to the host computer. Enclosed by the dotted line is the hierarchical exchanging network, where small circles are bidirectional bus switches. The network is a binary tree on which processor modules are connected as leaves (large circles in Fig.10). Though the HYPHEN system has a hierarchical structure, all the processor modules are uniformly leaf nodes. The access cost is the lowest between the nearest neighbors and becomes greater as they are getting apart from each other.

Each processor module consists of a processor unit and a memory unit, and has its own address which uniquely specifies the module in the HYPHEN system. Table 1 shows the detail of a processor module of the current HYPHEN C-16 system.

The host computer, which is under the Unix† (System-III) operating system on a Motorola MC68000, is connected to the processor No.0 with a serial communication line. Communicating with the processor No.0, the host computer behaves itself like one of processor modules in HYPHEN C-16.

---

†Unix is a registerd trademark of AT&T Bell laboratory.

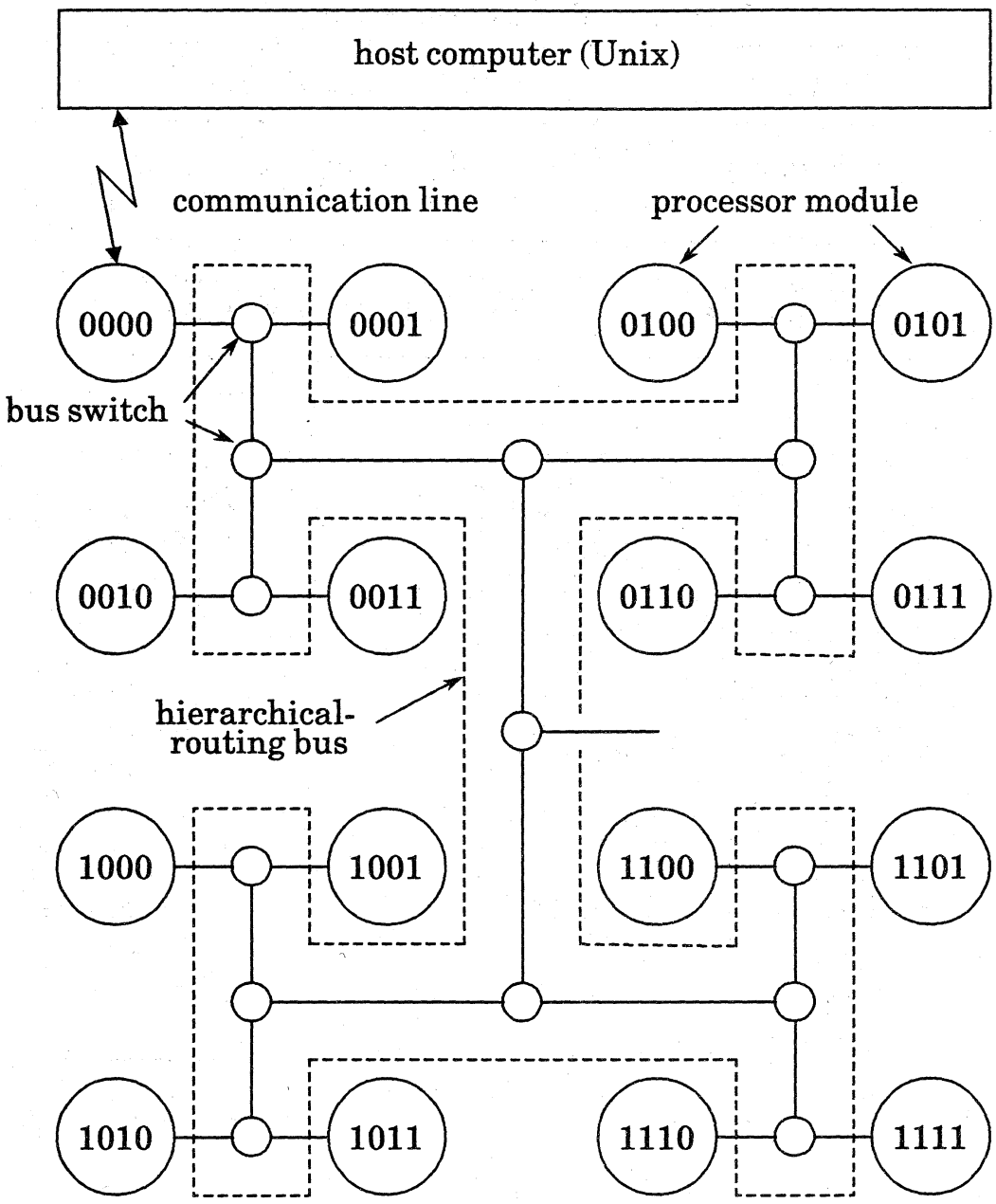

Fig.10 Configuration of HYPHEN C-16

4.2 Processing System of Nano-2

Fig.11 shows a flow of the processing system of Nano-2. A source program, which may consist of several files separately written in Nano-2, is compiled and then linked with system libraries to obtain the executable program. The loader allocates and distributes every partition of the program into a physical processor on HYPHEN C-16. Most of processing in Fig.11 are done on the host.

Table 1 Processor module

CPU	Z-80A (clock 4MHz)
Memory	ROM 4K bytes RAM 12K bytes
Communication Interface	Serial I/O 2 channels Parallel I/O 1 channel

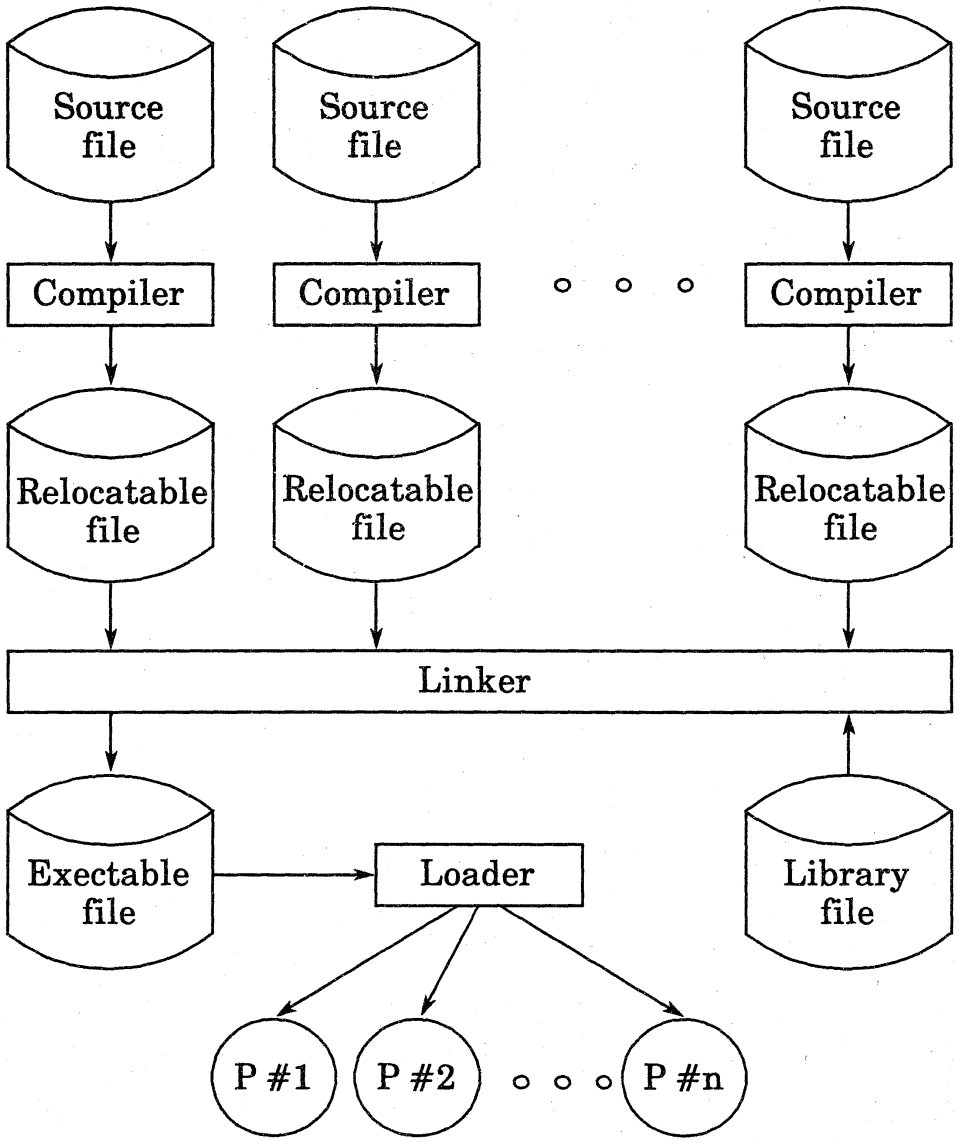

Fig.11 Flow of the processing system of Nano-2

—Compiler—

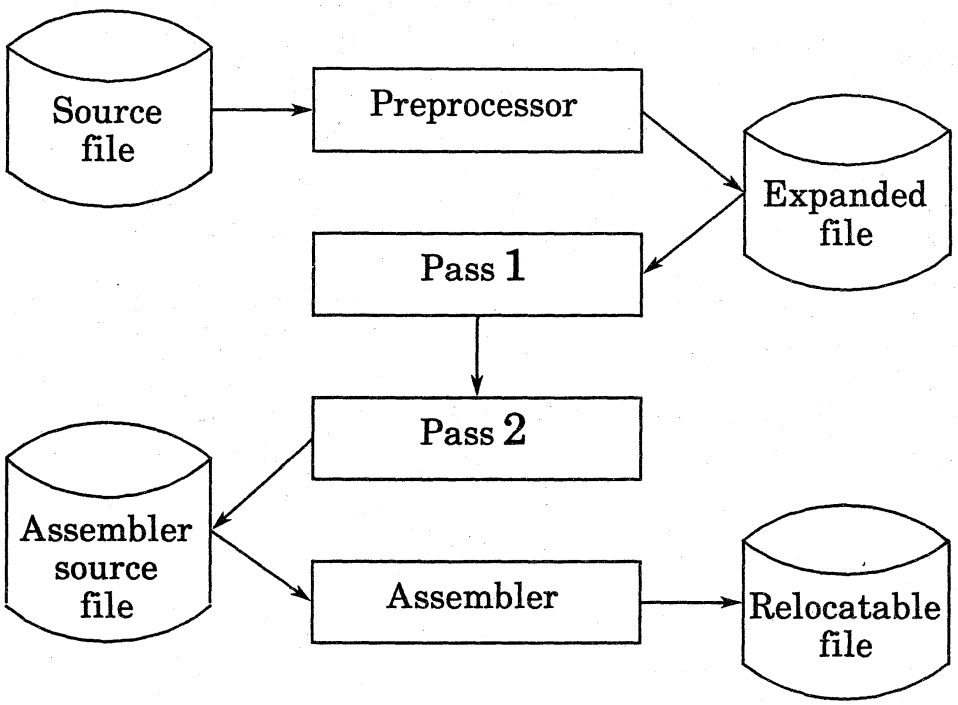

FIg.12 Construction of the Nano-2 compiler

Fig.12 shows the construction of the Nano-2 compiler. The function of each pass is explained below.

(1)preprocessor—treats including files and expanding macros. This is currently implemented as an extended version of cpp (the preprocessor of C compiler). It also includes the facility of repetitive compilation (mentioned in section 3.2). After module arrays were implemented, this facility is only used in the case that the number of repetitions is small and high-speed execution is required.

(2)pass 1—analyzes the syntax and points out most of syntactic errors. This pass is, for the most part, written in so-called compiler-compiler languages and has capability of recovery from syntactic errors. The lexical analyzer is written in lex provided by Unix, while the syntax analyzer is written in a yacc-like language which was implemented to construct the Nano-2 compiler.

(3)pass 2—analyzes the semantics and generates the object code. Strict checks of type, visibility and accessibility are done in this pass. The

object code generated is a calling sequence for run-time routines, called *threaded code*[6].

(4)assembler—converts an input assembly program into relocatable format.

—Linker—

The linker mainly performs the following two functions:

(1)to check consistency of external references between separated programs, and

(2)to combine a normal module and all floating constructs referred from it into a loading unit executed on a logical processor.

Before Nano-2 was implemented, the linker had allocated loading units to physical processors according to a specification by a user (, which is currently deferred until loading-time as described below).

—Loader—

The loader carries out both processor allocation and program distribution. Allocation means mapping logical processors to physical processors. It is difficult to find out suitable allocation especially on systems with structured access mechanisms. For example, frequent accesses between physical processors No.0 and No.F (in hexadecimal) in Fig.10 is harmful for communications by others which take a part of the same access path. Without carefully considering it, the root node would easily become a bottleneck. Access contention is an important issue as well as balancing load. For the present, since locality of data accesses and load of execution can not be derived from object codes, the loader automatically determines allocation on the next assumptions.

(1)The hierarchical structure of a Nano-2 program implies locality of data accesses. Because modules are designed by fanctional decomposition, the amount of internal references will be in excess of one of external references.

(2)The total size of loading units of each processor implies load of execution. It is, however, an *ad hoc* assumption.

Little access contention and balanced load will improve run-time efficiency of a program.

Because these assumptions do not necessarily hold, a user may specify allocation to the loader as well in the following special cases. When the program module depends on the system configuration, such as processing input/output requests, it had better locate on the processor which possesses appropriate devices. To facilitate debugging, it is helpful to allocate all physical processors to a single (usually the host) processor. Of course it must be possible to allocate several logical processors to a physical processor. The program can be developed independently of the number of physical processors.

After determination of allocation, the loader distributes loading units into each physical processor according to their allocation. When loading units include some module arrays, broadcasting can be used because a single loading unit is copied into many processors. With software broadcasting on the hierarchical exchanging network, the order of loading time would be logN (N is the number of elements).

4.3 Run-time Environment

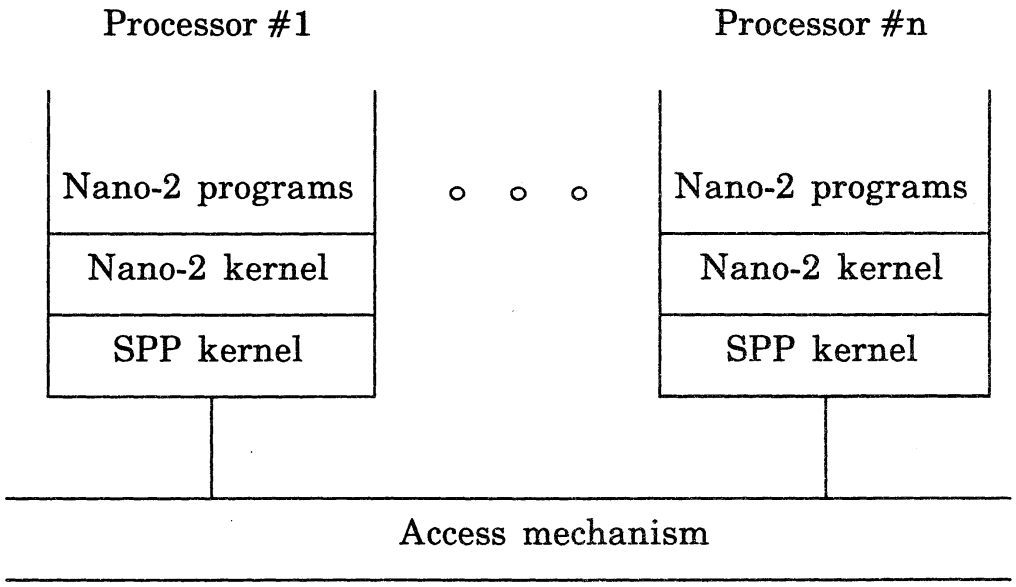

Fig.13 Run-time routines of each processor

Run-time routines, which supports execution of Nano-2 programs, are placed on every processor as shown in Fig.13. They consist of the following two layers.

—Nano-2 kernel—

The upper layer, called Nano-2 kernel, consists of routines for task scheduling, input/output requests and memory management. In the current implementation the compiler generates a calling sequence for run-time routines, called *threaded code*[6]. This approach facilitates the implementation with various kinds of processors. The system can commonly treat object codes. For example, the loader can distribute a certain object module into any processor without distinction of the cpu. In current HYPHEN C-16 (each processor module uses a Zilog Z-80), a processor module using an Intel 8086 and/or the host computer using a Motorola MC68000 can be indeed connected as one of the processors.

There are basically two methods to specify the location: one is local addressing inside a processor and the other is remote addressing between processors. These addressings are managed with access protections in the layer because the Z-80 does not possess the capacities of address relocation nor memory protection. The kernel covers this heterogeneous configuration and insufficient processor capacities at the expense of execution time.

As one of the run-time routines, the mechanism of parallel procedure call with reply is implemented with a countable semaphore. The current HYPHEN system only provides binary semaphore by using a hardware instruction, called *test-and-set*. Increment (or decrement) instructions executed in one memory cycle will be needed for the effective implementation of the mechanism. Parameter passing can be performed with no message buffering because the mechanism is basically a blocking type. By returning a reply on being invoked, it is, however, possible to simulate a non-blocking type.

—SPP kernel—

The lower layer called SPP kernel deals with invocations of routines and accesses to data on the other processors. This kernel converts requests, such as read/write functions which directly specify a

location on another processor, into message exchanging between both the processors. Although every processor is interconnected by serial communication interfaces on HYPHEN C-16, the upper layer can regard this loosely coupled system as a shared memory multiprocessor, which owes to functions of the SPP kernel.

This layer provides a minimum of functions so that it can be easily replace by hardware[5].

5. Conclusion

We have described the revised version of Nano-2 and its implementation. By introducing parallel procedure call with reply, module construct and its array, monitor, etc., we have removed particular concepts and notations of the HYPHEN system from the previous version. The strategy based on the shared memory model makes it easy to implement a parallel programming language on a multiprocessor even without memory sharing.

Module arrays improves efficiency of processing as well as writability and readability. For example, in Fig.8 of section 3.2, when we run the program with 120 elements (which is the number of primes and also the maximum size in current HYPHEN C-16), the amount of processing time and space of the compiler and the linker decreases to about 1/100. With broadcasting, loading time also decreases to about 1/15.

Automatic allocation helps beginners and omits a work in the case of simple parallel programs. In the complicated cases, because the allocation is hardly determined before execution, re-allocation cost is required to be low. This requirement is met in this system by determining allocation when loading-time.

Current HYPHEN C-16 has no memory sharing. But with shared memory model it is easy to implement and to replace the lower layer by hardware in real shared memory multiprocessors.

We regard the parallel computer HYPHEN and language Nano-2 as mere tools for investigation of parallel programming or distributed processing. A distributed operating system and a debugger for a parallel program are essential and must be implemented with Nano-2. We have a primary object in constructing parallel programming

environment on the HYPHEN system and studying parallelism from practical viewpoint.

Acknowledgment

We would like to thank Professor K.Ushijima for the useful suggestions about this research, and also thank Miss K.Kai for her helps in the improvement of the manuscript.

References

[1] Ada Programming Language (ANSI/MIL-STD-1815A), U.S. Government, Department of Defense, Ada Joint Program Office, 1983.

[2] Araki,K. and Arita,I.: "Compiler Control Facility As a Tool in Parallel Programming," Proc. 28th Annual Convention IPSJ, 1984.

[3] Araki,K., Arita,I. and Hirabaru,M.: "NANO-2: High-level Parallel Programming Language for Multiprocessor System HYPHEN," Proc, of COMPCON Fall'84, pp.449-456, 1984.

[4] Arita,I.: "On a Parallel Program with Synchronizing Mechanism Using FIFO Queue (I) —Self Synchronizing Parallel Program —," Trans. IPSJ, Vol.24, No.2, pp.221-229, 1983 (in Japanese).

[5] Arita,I. and Sueyoshi,T.: "On a Parallel Program with Synchronizing Mechanism Using FIFO Queue (III) —Execution Control Mechanism —," Trans. IPSJ, Vol.24, No.6, pp.838-846, 1983 (in Japanese).

[6] Bell,J.R.: "Threaded Code," Commun. ACM, Vol.16, No.6, pp.370-372, 1973.

[7] Brinch Hansen,P.: "The Architecture of Concurrent Programs," Prentice -Hall, 1977.

[8] Dijkstra,E.W.: "Guarded Commands, Non-determinancy and Formal Derivation of Programs," Commun. ACM, Vol.18, pp.453-457, 1975.

[9] Hirabaru,M., Araki,K. and Arita,I.: "Implementation of Parallel Programming Language P-Pascal," Technical Rep. of IPSJ, SF9-3, 1984 (in Japanese).

[10] Hirabaru,M., Araki,K., Sueyoshi,T. and Arita,I.: " Implementation of High-level Parallel Programming Language Nano-2," IECEJ, EC85-11, pp.35-46, 1985 (in Japanese).

[11] Hoare,C.A.R.: "Monitors: An Operating System Structuring Concept," Commun. ACM, Vol.17, No.10, pp.549-557, 1974.

[12] Holt,R.: "Concurrent Euclid, the UNIX System, and TUNIS," Addison-Wesley, 1983.

[13] Lampson,B.W., Horning,J.J., London,R.L., Mitchell,J.G. and Popek,G.L.: "Report on the Programming Language Euclid," SIGPLAN Notices, Vol.12, No.2, 1977.

[14] Mao,T.W. and Yeh,R.T.: "Communication Port: A Language for Concurrent Programming," IEEE trans., SE-6, pp.194-204, No.2, 1980.

[15] Sueyoshi,T., Saisho,K. and Arita,I.: "HYPHEN C-16 ——A Prototype of Hierarchical Highly Parallel Processing System," Trans. IPSJ, Vol.25, No.5, pp.813-822, 1984 (in Japanese).

[16] Sueyoshi,T., Arita,I., Saisho,K. and Ushijima,K.: "Programming System for Performance Evaluation of MIMD Type Parallel Computer HYPHEN C-16," Trans. IPSJ, Vol.26, No.6, pp.1097-1105, 1985 (in Japanese).

[17] Wirth,N.: "Modula: A Language for Modular Multiprogramming," Software-Practice and Experience, Vol.7, No.1, pp.3-35, 1977.